To Overlap or Not to Overlap: Optimizing Incremental MapReduce Computations for On-Demand Data Upload

Stefan Ene, Bogdan Nicolae, Alexandru Costan, Gabriel Antoniu

To Overlap or Not to Overlap: Optimizing Incremental MapReduce Computations for On-Demand Data Upload

Stefan Ene
University Politehnica
Bucharest
Bucharest, Romania
stefan.ene@cti.pub.ro

Bogdan Nicolae
IBM Research
Damastown, Mulhuddart
Dublin, Ireland
bogdan.nicolae@acm.org

Alexandru Costan
IRISA / INSA de Rennes
Campus de Beaulieu
Rennes, France
alexandru.costan@inria.fr

Gabriel Antoniu
Inria Rennes
Campus de Beaulieu
Rennes, France
gabriel.antoniu@inria.fr

ABSTRACT

Research on cloud-based Big Data analytics has focused so far on optimizing the performance and cost-effectiveness of the computations, while largely neglecting an important aspect: users need to upload massive datasets on clouds for their computations. This paper studies the problem of running MapReduce applications when considering the simultaneous optimization of performance and cost of both the data upload and its corresponding computation taken together. We analyze the feasibility of incremental MapReduce approaches to advance the computation as much as possible during the data upload by using already transferred data to calculate intermediate results. Our key finding shows that overlapping the transfer time with as many incremental computations as possible is not always efficient: a better solution is to wait for enough to fill the computational capacity of the MapReduce cluster. Results show significant performance and cost reduction compared with state-of-the-art solutions that leverage incremental computations in a naive fashion.

Keywords

MapReduce, data management, incremental processing

1. INTRODUCTION

In recent years, big data analytics has proven an indispensable tool in transforming science, engineering, medicine, healthcare, finance and ultimately business itself, thanks to the unprecedented ability to extract new knowledge and automatically find correlations in massive datasets that naturally accumulate in our digital age [10]. In this context, the MapReduce [7] model and its open-source implementation, Hadoop [20], were widely adopted by both industry and academia, thanks to a simple yet powerful programming model that accelerates the application development while inherently enabling a high throughput and scalability for processing large volumes of data.

However, an important limitation of the original MapReduce model is the constraint of having all data available before a job can be started. This limitation can become a serious problem especially when the input data is massive and needs to be uploaded from an external source, which involves large data transfers over network links of limited capacity: under such circumstances, even the best MapReduce implementation cannot stop the overall time-to-solution from growing to unacceptable levels. Still, due to its perceived simplicity, the most widely used approach encountered in practice does consist in uploading all necessary data to a MapReduce cluster before running the corresponding MapReduce job.

To cope with the challenges posed by the need to process massive input data, a possibly better solution would consist in overlapping data transfer with data processing. Although apparently straightforward, such a solution is not trivial, as it requires the ability to perform efficient incremental computations, i.e., leverage older results when new data is added to an existing dataset in order to save time in the calculation of the new overall result. In this case, all the time required to perform the data transfers can be leveraged to calculate intermediate results that simplify the computation once the data transfer is finished. In the context of MapReduce, one way to apply this idea is to modify the application to exhibit incremental behavior. This however is difficult and undesired, both because of added complexity and of performance issues (e.g., because intermediate results need to persist in the storage layer).

Luckily, a whole family of extensions to MapReduce and its implementations have recently emerged in order to facilitate efficient incremental computations (e.g. *Hourglass* [9]). However, such extensions are mostly designed for a scenario when the amount of data is unknown in advance, but intermediate results are desired at known moments in time (e.g., daily statistics). In our case, the exact opposite is true: the total amount of data is known in advance and the moment when intermediate results are generated is not important: they are only needed as long as they help reduce the complexity of the computation once the data transfer is finished.

This novel perspective on incremental MapReduce computations is further complicated by the rise of cloud computing [19]: since most users cannot afford buying and maintaining their own big data infrastructure, they often prefer the pay-as-you go utility model of cloud computing where the costs are proportional to the amount of computational resources used by the application. In this context, it is not enough to minimize the time-to-solution (which implies the need to minimize the compute time after the data trans-

fers have finished); in addition, another important goal is to *reduce the overall costs* required to obtain the solution!

In this paper, we explore how to optimize incremental MapReduce computations specifically for the case when a large dataset needs to be transferred and processed on a remote cloud. In a nutshell, our approach advances the computation during the data transfers as much as possible, but with the minimal possible effort. This improves the time-to-solution while minimizing the cost necessary to do so. To this end, we show that it is not always efficient to attempt to overlap the transfer time with as many incremental computations as possible, but rather a trade-off is involved: sometimes it is better to wait for more data to accumulate before starting any incremental computation.

We summarize our contributions as follows:

- We introduce a series of design principles that facilitate efficient incremental MapReduce computations during ongoing data transfers. In particular, we establish a correlation between the capacity of the compute cluster and the amount of incremental data to be processed. It serves to decide when to initiate an incremental computation and when to wait for more data to accumulate (Section 3.1).
- We show how to materialize these design principles in practice through a series of algorithmic descriptions (Section 3.2) illustrated by a prototype architecture (Section 4) and implemented in practice using the *Hourglass* [9] framework (Section 5).
- We evaluate our approach in a series of extensive experiments conducted on the Grid'5000 [4] testbed, using a real-life MapReduce application. They demonstrate significant benefits for our approach in terms of cost and compute time after the data transfer has finished, both when compared to a simple incremental approach and to a serialized approach that waits until the data transfer has finished (Section 6).

2. RELATED WORK

The MapReduce model and its implementations have been recently extended to better fit a variety of application requirements. Examples of such extensions include interactive analytics, job pipelining, iterative processing and the support for incremental computations. We focus on this latter direction. Previously proposed frameworks try to cope with incremental workloads in two ways: 1) by reusing the results of prior computations transparently through iterative processing, at the cost of additional overhead brought by loop control mechanisms; 2) by introducing new programming models to store and use state across successive runs so that only the necessary sub-computations need be performed. In contrast, our approach maintains the clean programming model of MapReduce and does not add any component for centralized control.

A class of research efforts leverage *iterative frameworks* for incremental processing, attempting to reuse the results of previous computations and targeting applications that reuse a working set of data across multiple parallel operations. Spark [21] was developed to optimize iterative and interactive computations. It uses caching techniques to improve performance for repeated operations. Spark exposes a functional programming interface that can be used interactively as a general-purpose language to process large datasets on a cluster. Rather than proposing new high-level programming languages, our optimizations rely on the simple MapReduce model, enhanced to support the overlap of transfers and computations, so that they can be easily adopted by the existing applications without any modification.

Several systems introduced support for *iterative MapReduce* processing. As Hadoop does not support iterative processing by design, HaLoop [5] was built on top of it with this purpose. It relies on a loop-aware task scheduler and on loop-invariant data caching. Besides HaLoop, other solutions accelerate iterative algorithms by maintaining iteration state in memory. Twister [8] employs a light-weight MapReduce runtime system and uses publish / subscribe messaging-based communication instead of a distributed file system. iMapReduce [22] and MapIterativeReduce [17] extract common features of iterative algorithms and provide support for them. All of these frameworks target applications with iterations across MapReduce jobs and require additional components and programming efforts to automatically aggregate their input and output. We specifically addresses this issue by scheduling map jobs as soon as their input data is available; map and data transfer jobs can thus be interleaved and the usual barrier between these two phases be eliminated.

More recently, dedicated solutions were proposed for *incremental processing*. Using basic arithmetic, they can update the output from the previous computations by adding and subtracting input data, instead of re-executing the whole processing. Hourglass [9] is developed at LinkedIn and targets computations over sliding windows. For these types of computations, the input data is partitioned in some way, usually according to time, and the range of input data to process is adjusted as new data arrives. It runs on unmodified Hadoop and provides an accumulator-based interface for programmers to store and use state across successive runs. In the database world, Google's Percolator [14] allows transactional updates to a database through a trigger-based mechanism. Similarly to Hourglass, the system provides a metaphor to keep track of the state of the incremental computation, called observer: piece of code that is invoked by the system whenever a user-specified column changes. The main drawback of these systems is their focus on particular use cases, which makes them difficult to get adopted by a larger, scientific community: their were written for some precise goals (e.g. dashboard statistics, web index updating) and incorporate some of the application semantics into the programming model (e.g. in Hourglass, the input data needs to be portioned by days).

In the *cloud data management* ecosystem, a popular solution is upload data to generic cloud storage services (e.g., Amazon S3 [1], Azure Blobs [6], etc.), after which the computations are started. Typically, they are not concerned by achieving high throughput, nor by potential optimizations, not to mention their inability to overlap transfers and computations. A number of alternative solutions aim to improve the cloud data transfers throughput by exploiting the network and the end-system parallelism or a hybrid approach between them. Multi-hop path splitting solutions [18] replace a direct TCP connection between the source and destination by a multi-hop chain through some intermediate nodes. Multi-pathing [15] employs multiple independent routes to simultaneously transfer disjoint chunks of a file to

its destination. These solutions come at some costs: under heavy load, per-packet latency may increase due to timeouts while more memory is needed for the receive buffers.

Due to a clear separation between the the data staging phase and the computation, the state-of-art approaches presented before focus on either one or another of the phases. Thus, to our best knowledge, *this is the first approach that simultaneously focuses on both data upload and processing.* By keeping resources idle in order to have "enough" data, our approach has the potential to drastically reduce costs in the context of the next generation Resource-as-a-Service clouds [3]. With this emerging model, compute and I/O resources will be rented and charged for in fine-grain, dynamically changing amounts and not in fixed bundles, as on today's IaaS clouds. Our proposal aims to become a building block of this new paradigm.

3. SYSTEM DESIGN

This section describes the design principles and algorithms behind our approach (Sections 3.1 and 3.2), while the next ones show how to apply them in a cloud architecture (Section 4) and finally how to efficiently implement them in practice (Section 5).

3.1 Design Principles

Our proposal relies on the following key ideas:

Advance the computation during data transfers using asynchronous incremental jobs.

One of our main goals is to minimize the computation time after the data transfer has finished, which ultimately minimizes the time-to-solution for our problem. To achieve this, we propose to advance the computation as much as possible during the data transfer by computing intermediate results asynchronously in order to leverage them after the transfer has finished. This is an iterative process: when a certain amount of data was collected, a new incremental job is started to obtain the result for the data received so far. While the job is running, new data is accumulated in the background. When the job has finished and enough new data is accumulated, a new incremental job is started that computes the new result and combines it with the previous result to obtain the overall result. The process is repeated as long as the data transfer is not finished or there is still unconsumed data left. Using this approach, the complexity of the problem at the moment when the data transfer has finished is reduced to the complexity of the last incremental job that needs to process the remaining data.

Wait to accumulate enough data to fill the computational capacity ("wave size").

A naive implementation of the iterative process described above is to start a new incremental job as soon as the previous one has finished, regardless of the amount of accumulated data. Such a solution is optimal when the completion time for a jobs is equal to the sum of the completion times for an arbitrary decomposition into sub-jobs. However, in our context this is hardly the case, because MapReduce jobs are highly parallelizable. To illustrate this point, consider the completion time for a trivial job that involves a single map task and its corresponding reducer. If we were to double the size of the problem, we would need two mappers that generate two times more output. However, if we have enough resources to run these mappers in parallel and use two reducers to handle the aggregation in parallel as well, then the completion time should remain virtually unchanged (assuming there is an even distribution of the values among the reducers). Starting from this observation, an intuitive generalization is that the completion time for an incremental job remains roughly the same as long as the data involved fits into the computational capacity of the MapReduce cluster (henceforth called *wave size*). Based on this intuitive generalization, it is easy to notice that starting an incremental job before wave size was accumulated in the background is sub-optimal, because it increases the compute time unnecessarily. This has two negative consequences: (1) there is an obvious risk to increase the time-to-solution (which is not mandatory if the computation can be masked by the overlap with the data transfer); and (2) a longer compute time means more operational costs due to resource utilization. Thus, an obvious conclusion is to wait until enough wave size data was accumulated in the background, such as to leverage the entire computational capacity at its full potential. Note that overhead of reading and combining the previously accumulated result with the output of the latest incremental job is handled during the reduce phase of the latest incremental job. Thus, it is not necessary to explicitly dedicate computational resources for this task. However, in a typical MapReduce setup, it is necessary to reserve a number of extra resources for the purpose of running speculative tasks towards the end of the job (e.g., in order to deal with slow mappers). Thus, for practical purposes, *wave size* is slightly smaller than the total computational capacity.

Optimize the number and size of increments according to wave size.

Although it may happen that the total size of the data to be uploaded is a multiple of wave size, for the vast majority of cases this does not hold. Thus, if we were to wait in the beginning until the wave size is reached before starting the first incremental job, we would be left in the end with less than wave size for the last job. However, since the last job takes the same time to finish regardless of the remainder size, a much better strategy is to wait in the beginning only until the remainder is reached, such that a full wave size remains for the last incremental job. Using this strategy, the overall time-to-solution is reduced, because starting the first incremental job with less data means less wait time in the beginning, that cannot be overlapped with the computation. Furthermore, once the first incremental job was launched and the rest of the data is guaranteed to be a multiple of wave size, it is possible to receive more than twice the wave size before a concurrent job has finished. If that is the case, we propose to set the increment size for the next job as the largest multiple of wave size smaller than the size of the newly received data. Using this optimization, we minimize the number of "fully loaded" incremental jobs necessary to process the difference, which avoids the unnecessary overhead resulting from running multiple successive incremental jobs (as exhibited by a naive approach that simply fixes the increment size to the wave size).

3.2 Algorithms

In this section, we zoom on the design principles presented in Section 3.1 by providing an algorithmic description. We

ignore the data transfer itself and assume that it is an external process initiated by the user that is completely decoupled from the computation and satisfies two properties: (1) the total size of the data to be transferred to the cloud storage (denoted D) is known in advance; and (2) there is a mechanism to query how much data was transferred so far (primitive QueryProgress). Furthermore, we assume that the wave size (denoted W) is proportional to the number of slots allocated for mappers and reducers (denoted N), minus a small number of reserved slots K (configurable value) used for running speculative mappers. More specifically, $W = C \cdot (N - K)$, where C denotes the chunk size (typically 64 MB).

Algorithm 1 Algorithm to minimize the number of increments and align them to wave size

```
procedure OptimizedIncComputation(D, N, K, C)
    W ← C · (N − K)
    wait until QueryProgress() ≥ D mod W
    Head ← ⌊ QueryProgress() /W⌋ + D mod W
    Result ← AsyncIncrementJob(0..Head, ∅)
    while Head < D do
        wait until AsyncIncrementJob done
        wait until QueryProgress() −Head ≥ W
        Diff ← ⌊(QueryProgress() −Head)/W⌋
        PrevHead ← Head
        Head ← Head + Diff
        Result ← AsyncIncrementJob(PrevHead..Head, Result)
    end while
    wait until AsyncIncrementJob done
end procedure
```

Algorithm 1 illustrates how to materialize the design principles presented in Section 3.1. Given D, N, K and C as input, the first step is to calculate W. Then, the next step is to wait until at least D **mod** W data was transferred. If more than D **mod** W was transferred, then it makes sense to enlarge the increment beyond D **mod** W such that it packs together as many full waves as possible beyond the remainder. Once the optimal size for the first increment was established, the corresponding asynchronous incremental job is started. For simplicity, we assume that the transferred data is streamed and stored as single unstructured sequence of bytes in the storage layer of the MapReduce deployment (e.g., a file in HDFS [2] or GPFS [16]). By convention, we delimit the data corresponding to an increment as $X..Y$, where X is the offset of the first byte and Y is the offset of the last byte in the sequence. However, this is only a simplified conceptual structuring that is by no means a requirement: in practice, the data corresponding to $X..Y$ may be stored in multiple files or even be part of a higher level abstraction (e.g. sharded and indexed in a custom fashion). The asynchronous incremental job is responsible both to process the data $X..Y$ and to combine the result with the previous intermediate result (which initially is $\emptyset$). The new intermediate result is stored into *Result*. Note that *Result* needs not be stored into the storage layer: this detail is left to the implementation of the incremental MapReduce middleware, which may choose to use a different location optimized for non-persistent data (e.g. local storage devices where intermediate mapper output is written).

Once the first increment was processed, the remainder $Head..D$ is processed in increments that span as many full waves as it was possible to transfer while AsyncIncrementJob was running. Note that since $Head..D$ is a multiple of W, the loop will always finish. Finally, a final wait is necessary for the last incremental job to finish, after which the algorithm completes.

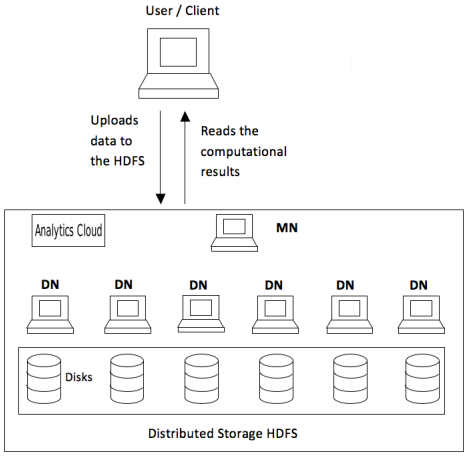


Figure 1: A simplified cloud architecture that integrates our approach

4. ARCHITECTURE

We depict a simplified cloud architecture that integrates our approach in Figure 1. The *cloud client* holds a large dataset that needs to be processed using a MapReduce job. It has access to an *analytics cloud* that employs a MapReduce deployment capable of supporting incremental computations. There are no constraints on the incremental MapReduce deployment itself: it can either be a custom installation on top of virtual machines in an IaaS cloud or a provider-managed PaaS. The only requirement is that the MapReduce deployment exposes its storage layer (e.g. HDFS [2] or GPFS [16]) externally to the cloud client, so that there is a direct way to upload the dataset while running the incremental MapReduce jobs at the same time.

Typically, the storage layer will comprise a set of data nodes (denoted DN), which are used at the same time as compute nodes for the MapReduce jobs. Thanks to this architectural feature of MapReduce deployments, the data nodes will mostly serve local I/O requests (e.g. issued by mappers scheduled close to the data), which minimizes the network bandwidth utilization and thus any potential interference with the data upload. Thus, exposing the storage layer to the user is not only efficient because it enables direct uploads, but also because it is well suited by design for our scenario.

The algorithm presented in Section 3.2 is implemented as a *runtime control module* that runs on the master node (denoted MN) and is responsible to coordinate the data transfer and initiate the incremental MapReduce jobs. It can be deployed either by the user directly, or it can be exposed by the cloud provider as a service in itself. Thanks to the decoupling of the data transfer from the computation, there is no constraint with respect to when or in what order to start the two. If the computation starts before the data transfer, then it will wait until D **mod** W data was gathered.

If it starts after the data transfer, the initial optimization (that packs together as many full waves as possible beyond the reminder for the first incremental job) can efficiently handle even a scenario where a large part of the data transfer is already complete. Even at the extreme, when the data transfer is already finished before the computation is started, our algorithm gracefully reduces the computation to a single job, without splitting the computation unnecessary into multiple increments. Thus, users can take advantage of this flexibility to optimize when to start the data transfer and the computation, depending the dynamicity of cost model (e.g., delay until the price of the bandwidth utilization or the VM instances reaches a desired level).

5. IMPLEMENTATION

We have chosen to implement our approach on top of *Hourglass* [9], an incremental MapReduce framework specifically designed to make computations over sliding windows. These sliding windows typically correspond to temporal data partitioned on a per-day basis and are particularly useful in two scenarios: (1) the length of the window is set to some constant number of days and the entire window moves forward as new data becomes available (e.g., a daily report summarizing the the number of visitors to a site from the past 30 days); and (2) the beginning of the window stays constant, but the end slides forward as new input data becomes available (e.g., a daily report summarizing all visitors to a site since the site launched).

In our case, the second scenario applies: the window represents the data transferred so far, for which an updated result is needed. Since Hourglass requires the input data to be partitioned on a per-day basis, we decided to associate the notion of day to chunk size, such that the smallest possible increment that can be processed by an incremental job is a chunk. This level of granularity is also reported by the QueryProgress() primitive used in Algorithm 1 (i.e. the data transfer is reported to have made progress only when at least a full chunk was transferred to the storage layer).

The runtime control module (mentioned in Section 4) that schedules the incremental MapReduce jobs is implemented as a series of bash scripts. Besides the functionality corresponding to Algorithm 1, the implementation also reorganizes the data in such way that it fits the per-day partitioning layout expected by Hourglass. More specifically, Hourglass being implemented on top of Hadoop, it relies on HDFS as its storage layer and expects a fixed directory and file structure for the days. This reorganization of data happens on the client side, before it is uploaded remotely to the HDFS deployment.

6. EVALUATION

This section presents a preliminary experimental evaluation of our work using a real-life MapReduce application.

6.1 Experimental setup

The experiments were performed on the Grid5000 [4], an experimental testbed that federates nine sites in France. We used 11 nodes (interconnected with Gigabit Ethernet) of the Rennes site, each of which is equipped with two quad-core AMD Opteron 6164 HE 1.7 Ghz processors, 48 GB memory and local storage of 250 GB (SATA AHCI Controller).

Out of the 11 nodes, 10 nodes are used for the Hourglass deployment and the remaining node corresponds to the client and is used as the source of data upload. We use Hourglass v0.1.3 (based on Hadoop v1.2.1) on the 10 nodes in the following configuration: all nodes serve the role of slaves, running a co-located TaskTracker and HDFS DataNode process using 12 slots for mappers and reducers. In addition, one node also serves the role of master, running a JobTracker and a HDFS NameNode (for a small deployment such as ours, the management overhead incurred by the master role is small enough to justify co-location with the slave role). The chunk size is fixed at 64 MB. Thus, the total capacity of the cluster is $N = 120$ slots, out of which $K = 10$ are reserved, resulting in a wave size $W = 7040$ MB. Furthermore, the runtime control module that implements our approach (described in Section 5) is also running on the master node.

An important part of our study is to understand how the computation and the data transfer can be overlapped when using different upload speeds. This aspect is important, because cloud clients may have different upload capabilities depending on network quality or even cost (e.g., they would prefer to use a low upload throughput in order to pay less for bandwidth utilization). To this end, we use Traffic Control (TC) to limit the outgoing traffic from the client node to a desired throughput. For our setup, we achieved a maximum write throughput into HDFS of 17.4 MB/s (using the standard replication factor of three). Thus, in our experiments we vary the upload throughput from 1 MB/s to 17.4 MB/s.

The computation itself is a real-life MapReduce application that counts the number of events for a given id and is part of the Hourglass distribution. It is conceptually similar to WordCount, but has different input format: a big text composed of lines, with each line corresponding to an event (which is frequently the case of log files). The input data for this application is (unless otherwise specified) 20 GB and was generated using a random generator that is part of the Hourglass distribution. This results in a minimum of three waves necessary to complete the computation.

6.2 Methodology

We compare three approaches throughout or evaluation:

Serialization of upload and computation.

This approach is non-incremental: first, the dataset is uploaded into HDFS. Once the upload has finished, the computation is performed using a single job that covers the whole dataset. For the rest of this paper, we refer to this approach as simple.

Start next incremental job as soon as the previous one finished.

This approach implements a naive incremental strategy that starts the computation right away, waiting only for a minimal amount of data to accumulate for the first increment. Then, it accumulates new data in the background while the incremental job is running and starts the next incremental job as soon as the previous one finished. For the rest of this paper, we refer to this approach as greedy.

Our approach.

This approach minimizes the number of increments and optimizes their size to be as close as possible to a multiple of computational capacity, according to the algorithm dis-

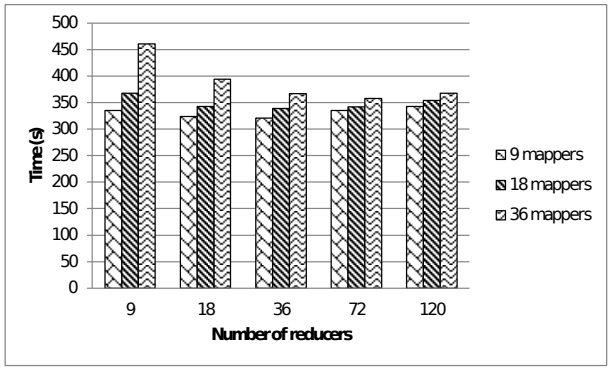


Figure 2: Weak scalability of Hourglass jobs: runtime for an increasing dataset size (expressed as number of parallel mappers needed to process it) using a variable number of reducers. Each mapper processes a chunk (64 MB).

cussed in Section 3.2. For the rest of this paper, we refer to this approach as our−approach.

These three approaches are compared based on the following metrics:

- **Time-to-solution** is the total time elapsed between the moment when the data transfer has started and the moment when the overall results are obtained. This metric represents the absolute performance as observed by the end user.
- **Compute time after data upload** is the time required to finalize the computation after the data upload has finished. This metric is important because it shows how well each of the approaches can overlap the computation with the data transfer in order to make progress during the data upload. A low value indicates better performance.
- **Cost efficiency** represents a cost estimation for running the application, using a simple cost model that differentiates between two states: (1) the MapReduce cluster is in stand-by, waiting for a new incremental job to be issued; (2) the MapReduce cluster is busy processing an incremental job. The cost estimation is $C_{approach} = W_c * T_c + W_i * T_i$, where W_c, W_i are coefficients corresponding to the cost perceived per unit of time when the cluster is busy/idle, while T_c, T_i is the duration of the two states. A lower value indicates lower cost.

6.3 Preliminaries

Before running the comparative evaluation, we present in this section a series of preliminary experiments that aim to demonstrate the fact that the assumptions we base our design principles upon (described in Section 3.1) hold in practice.

To this end, we focus on the evaluation of the weak scalability and strong scalability of the Hourglass application presented in Section 6.2. Figure 2, depicts the runtime for an increasing number of reducers corresponding to the dataset sizes of 0.57 GB, 1.15 GB, and 2.3 GB (which fill the capacity of 9, 18, 36 mappers respectively). Since the mappers are completely decoupled, the map phase is trivially scalable for an increasing number of mappers as long as the cluster capacity is not reached. Thus, the overall weak scalability of the job largely depends on the scalability of the reduce phase. As can be observed, when the number of mappers greatly outnumbers the number of reducers, there are not enough reducers to efficiently parallelize the reduce phase, which at the extreme leads to an increase of up to 35% in runtime (36 mappers and 9 reducers). However, when the number of reducers is increased to at least the number of mappers, the difference in runtime becomes less than 5% regardless of the data size. Furthermore, increasing the number of reducers beyond the number of mappers does not lead to significant differences in runtime. This can be explained by the fact that more reducers have a higher parallelization potential but suffer from higher overhead, because each reducer needs to collect a smaller piece from all outputs of the mappers (which are distributed). Thus, we conclude that using the maximum number of reducers and waiting for enough data to accumulate in order to fill the maximum number of mappers is enough to achieve a near-optimal solution. Ultimately, this confirms our intuition to use incremental jobs that process data sizes that are multiple of the wave size.

6.4 Performance evaluation

Our next series of experiments focuses on the performance of the three approaches described in Section 6.2. To this end, we evaluate each approach for a variable upload speed that is increased from 1.8 MB/s to 17.4 MB/s (which is the maximum throughout that the HDFS deployment can sustain). The upload speed is controlled according to the mechanism described in Section 6.1.

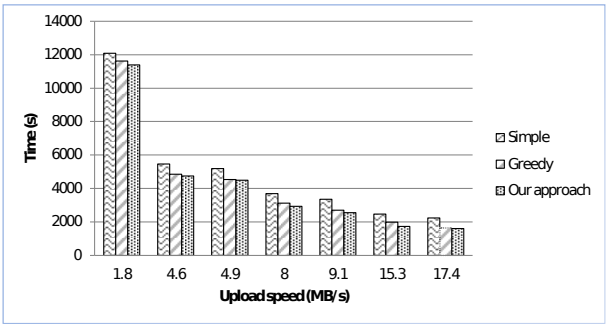
The results are depicted in Figure 3. The time-to-solution (depicted in Figure 3(a)) reveals an interesting finding: for small upload speeds, only comparatively small fraction of time is needed for the computation. Thus, all three approaches perform closely, with our−approach being faster than simple and greedy by 5% and 1% respectively. As the upload speed increases, the difference between simple and the other two approaches increases significantly, with our approach up to 30% faster at the extreme of 17.4 MB/s. However, the difference with respect to greedy remains still at 1%.

Although in absolute terms the difference between greedy and our−approach is modest, this is expected because the throughput of the data transfer is significantly smaller than the throughput of the computation. However, note that this is a consequence of the simplicity of the application we considered: in a more computationally-intensive scenario, the differences in performance are expected to grow accordingly in favor of our approach.

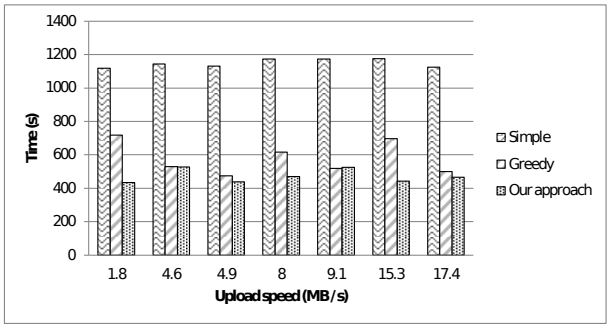
To understand this effect better, we focus in Figure 3(b) on the compute time necessary to achieve the end result *after the data upload has finished*. By eliminating the data upload from the analysis, the difference between the three approaches becomes significantly larger and highlights a much lower computational overhead for our−approach. As it can be observed, our−approach manages to reduce the compute time after the data upload has finished by up to 40% when compared to greedy (with no reduction in some cases) and between 50%-60% when compared to simple.

6.5 Cost estimation

In this section, we estimate the cost necessary to run



(a) Time-to-solution (lower is better)



(b) Time needed to finish the computation after the data upload is complete (lower is better)

Figure 3: Performance results for a dataset size of 20 GB and a variable upload speed. Wave size is 110 mapper/reducer slots and 10 slots are reserved.

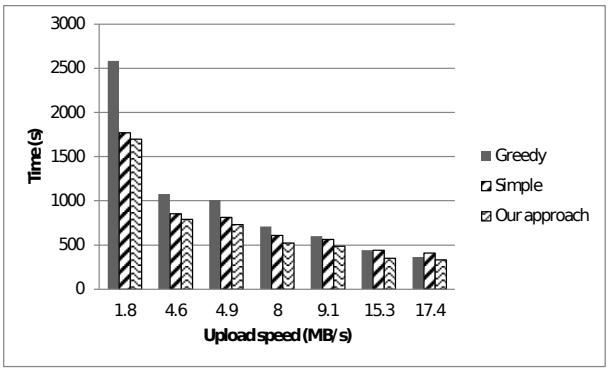


Figure 4: Cost estimation for a a dataset size of 20 GB and a variable upload speed. Wave size is 110 mapper/reducer slots and 10 slots are reserved.

the application according to the cost metric $C_{approach}$ introduced in Section 6.2. To this end, we monitored in the previous set of experiments (Section 6.4) for each approach and each upload speed how much time the Hourglass deployment spends in idle mode and how much time it is busy with running incremental jobs.

Based on these measurements, we calculated $C_{approach} = W_c * T_c + W_i * T_i$ for each node individually and summed up the results for all 10 nodes in Figure 4. The coefficients W_i and W_c were fixed at 50 and 80, based on previous observations that nodes on Grid5000 consume 50 Wh when idle and 80 Wh when running MapReduce computations. We decided to use coefficients that reflect energy consumption in the cost model, because this is representative of the minimal operational cost incurred by the application. More specifically, the cloud provider needs to charge its users at least this amount, otherwise its business would not be profitable. By using this lower bound, we compare the cost effectiveness of the three approaches *in the worst case*, which makes our results relevant no matter how much the real prices are driven down by competition and proposals to adopt finer-grain cost models [3] that bridge the gap between provisioned resources and their actual utilization.

As can be observed, our−approach incurs the least cost regardless of upload speed, both when compared with greedy and simple. The largest difference is observable for a slow upload speed: in the case of 1.8 MB/s, our approach reduces the cost by 44% compared with greedy. However, as the upload speed increases, the greedy approach becomes increasingly more cost-effective (down to 10%). On the other hand, the opposite holds for the difference between our−approach and simple: there is a significant growth from 4% in the case of 1.8 MB up to 23% in the case of 17.4 MB.

7. CONCLUSIONS

In this paper, we explore how to optimize incremental MapReduce computations specifically for the case when the input data is a large remote dataset that needs to be uploaded to the cloud first. Our main goal is to advance the computation as much as possible during the upload phase, such that the time spent in computations after the upload finishes is minimized. Furthermore, another important goal is to minimize the operational cost of doing so.

Unlike state of art approaches that are either optimized for different purposes or treat the computational problem independent of the data upload, to our best knowledge, *this is the first approach which simultaneously focuses on both data upload and processing*. In this context, we show that it is not always efficient to attempt to overlap the transfer time with as many incremental computations as possible: a better solution is to wait for enough to fill the computational capacity of the MapReduce cluster. Based on this idea, we developed and evaluated a preliminary prototype.

To demonstrate the viability of our prototype in real-life, we run extensive experiments in a distributed setting that involves a 11-node large incremental MapReduce deployment based on Hourglass. The results show significant benefits for our approach when compared with a simple incremental strategy that starts the next incremental job immediately after the previous has finished: the time-to-solution is improved by 1%, the compute time after the data transfer is finished is reduced by up to 40% and the cost is reduced 10%-44%. Compared with a serialized strategy that starts the computation only after all data was transferred, the time-to-solution is improved by up to 30%, the compute time after the upload finished is reduced by up to 60% and the cost is reduced between 4%-23%.

Encouraged by these results, we plan to broaden the scope of our work and experiment with more applications. We are

also looking at the possibility of dynamically adapting the wave size to account for fluctuations in processing time or uneven distribution of values among reducers. Finally, we considered the cost optimization aspect by making use of a simple cost model. Using a more complex cost model that takes fluctuations in the price of resources and I/O bandwidth utilization into consideration is a promising avenue as well. Furthermore, we focused only on the cost from the perspective of the computation. Thus, an interesting avenue is to explore the cost also when taking storage into consideration. In this context, we plan to study the viability of several storage elasticity features introduced by our previous work [11, 12, 13].

8. REFERENCES

[1] Amazon S3. http://aws.amazon.com/s3/.

[2] Apache HDFS. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[3] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The Rise of RaaS: The Resource-as-a-service Cloud. *Commun. ACM*, 57(7):76–84, July 2014.

[4] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *Int. J. High Perform. Comput. Appl.*, 20:481–494, November 2006.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[6] B. Calder and et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A Runtime for Iterative MapReduce. In *HPDC '10: The 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818, Chicago, USA, 2010.

[9] M. Hayes and S. Shah. Hourglass: A library for incremental processing on Hadoop. In *BigData'13: The 2013 IEEE International Conference on Big Data*, pages 742–752, Santa Clara, USA, Oct 2013.

[10] T. Hey, S. Tansley, and K. M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[11] B. Nicolae. Understanding Vertical Scalability of I/O Virtualization for MapReduce Workloads: Challenges and Opportunities. In *BigDataCloud'13: 2nd Workshop on Big Data Management in Clouds*, pages 3–12, Aachen, Germany, 2013.

[12] B. Nicolae, P. Riteau, and K. Keahey. Bursting the Cloud Data Bubble: Towards Transparent Storage Elasticity in IaaS Clouds. In *IPDPS '14: Proc. 28th IEEE International Parallel and Distributed Processing Symposium*, pages 135–144, Phoenix, USA, 2014.

[13] B. Nicolae, P. Riteau, and K. Keahey. Transparent Throughput Elasticity for IaaS Cloud Storage Using Guest-Side Block-Level Caching. In *UCC'14: 7th IEEE/ACM International Conference on Utility and Cloud Computing*, London, UK, 2014.

[14] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–15, 2010.

[15] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handley. Data center networking with multipath TCP. In *Hotnets-IX: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 10:1–10:6, Monterey, USA, 2010. ACM.

[16] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02: The 1st USENIX Conference on File and Storage Technologies*, Monterey, USA, 2002.

[17] R. Tudoran, A. Costan, and G. Antoniu. MapIterativeReduce: A Framework for Reduction-intensive Data Processing on Azure Clouds. In *MapReduce '12: The 3rd International Workshop on MapReduce and Its Applications Date*, pages 9–16, Delft, The Netherlands, 2012. ACM.

[18] R. Tudoran, A. Costan, R. Wang, L. Bouge, and G. Antoniu. Bridging Data in the Clouds: An Environment-Aware System for Geographically Distributed Data Transfers. In *CCGrid'14: The 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 92–101, Chicago, USA, 2014.

[19] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, Jan. 2009.

[20] T. White. *Hadoop: The Definitive Guide*. Yahoo! Press, USA, 2010.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud'10*, pages 10–10, Boston, USA, 2010. USENIX.

[22] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: A Distributed Computing Framework for Iterative Computation. In *IPDPSW'11: Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing*, pages 1112–1121, 2011.